第4章 ソフトのハード化でボトルネックを解消する

── Nios II C-to-Hardwareアクセラレーション・コンパイラを使いこなす

猪狩貴寛

ここでは，米国Altera社の「Nios II C-to-Hardware Acceleration Compiler」を取り上げる．これは，ANSI Cソフトウェア・コードから，ハード・ワイヤード論理のアクセラレータ回路を生成する合成ツールである．C言語のソフトウェア・コードのうち，ユーザが指定したファンクション（サブルーチン）をRTL（Register Transfer Level）の回路（VHDLまたはVerilog HDL）に変換する．Cソース・コードを書き換えなくても使えるが，より高い性能を得るためにはガイドラインで示された推奨コーディング・スタイルに従う必要がある．そこで本稿では，Cソース・コードの記述の違いによるアクセラレーション効果の差について，例を挙げながら説明する．（編集部）

今日の製品開発においては，早期の市場投入を実現するために，開発期間の短縮が求められます．しかも競争力を高めるためには，性能の向上は欠かせません．このため，システム設計においては，ソフトウェアとハードウェアの切り分けが悩みどころになっています．一般には，ボトルネックとなる処理をハードウェア化し，残りをソフトウェア処理にします．

本稿では，ソフト・マクロのCPUを実装するFPGAの設計を想定し，ソフトウェア処理部のボトルネックの解消方法の一例を紹介します．使用するCPUコアは，米国Altera社の「Nios II」です．また，Nios IIの開発ツールとして提供されている「Nios II C-to-Hardware（C2H）Acceleration Compiler」を活用します．

Nios II C2H Acceleration Compiler（以降「C2H」と呼ぶ）は，C関数をハードウェア・アクセラレータに変換する合成ツールです．ソフトウェア処理をハードウェア化するので，動作周波数を上げずに性能を向上することが期待できます．Nios IIの開発環境（Nios II IDE，Quartus II，MegaCore）があれば使用できます．

C関数のハードウェア化のための操作は，基本的にはマウスで右クリックを行うだけです．C関数をハードウェアに変換し，Avalon（Altera社の独自オンチップ・バス）に統合します．合成したハードウェアにアクセスする関数（ラッパ部）の生成やソフトウェアのビルドを行います．ただし，C2Hによるアクセラレーション効果は，Cソース・コードの記述のしかたによって異なります．

そこで本稿では，Cソース・コードの記述のしかたとC2Hによるアクセラレーション効果について，実際のコードを用いて説明します．Cソース・コードとしては，sobelフィルタと呼ばれる画像の水平微分（エッジ検出）を行うプログラムを用います．Nios IIの開発環境はver.7.0を使用します．

1．ボトルネックの抽出を行う

ソフトウェア処理におけるボトルネックの抽出は，ソフトウェア開発ツールの持つ機能で行えます．例えばGNUコンパイラ（gcc）を使用する場合は，オプション-pgを付けてコンパイルし，実行するだけです．生成されたモニタリング・ファイル（gmon.out）には，関数などの実行時間が示されているので，これで確認します．

Nios IIのソフトウェア開発用に用意されているコンパイ

KeyWord　FPGA，Nios II，C2H，ANSI C，ソフト・マクロ，CPUコア，Sobelフィルタ，エッジ検出

ラは，GNUコンパイラをベースとしています．従って，Nios II IDE上から，オプションの指定を行えばよいことになります．具体的には，Nios II IDE上からSystem Library（**図1**）を開き，「Link with profiling library」をチェックします．これにより，gccの-pgオプションが有効になります．

この後，Build（makeコマンド相当）を行い，実行すると**図2**のようにgmon.outが得られます．gmon.outのうち，time欄は，実行時の要した時間を割合（%）で示したものです．関数などに分けて示されます．

図2の例では，`special_filter`と`alt_irq_register`の二つの関数で全体の処理時間のほとんどを要していることが分かります．今回は`special_filter`をハードウェア化対象関数として扱います．

2. C2Hの制約とチューニングのポイント

C2HはANSI Cをサポートしています．ただし，ハードウェア化ツールのため，`printf`をはじめとする標準関数はサポートしていません．本稿執筆時の制約事項は以下の通りです．

- `printf`，`memcpy`などの標準関数
- ハードウェア化に適さない関数の再起呼び出し
- 浮動小数点演算（将来サポート予定，ペリフェラルとして実装することで可能）
- 関数からのサブ関数呼び出し

C2Hを効果的に活用するためには，以下で説明する各項目に対応する必要があります．実際の設計（チューニング）では，要求性能を満たす段階まで対応すればよく，すべての項目を実行する必要はないでしょう．

● ソース・コードの分離

デバッグ効率を上げるために，ソース・コードの分離を行います．

コンパイラは，ソース・ファイル単位でコンパイルを行います．そこで，ハードウェア化対象関数を別ファイルに分割しておきます．こうすることで，期待通りに動作しない場合に，問題がハードウェア側にあるのか，ソフトウェア側にあるのかの切り分けをしやすくなります．

また，C2Hにより合成されたハードウェアの信頼性を上げるため，ハードウェア化対象関数では，ソフトウェアの段階でワーニングをなくしておきます．

今回の例では，ハードウェア化対象関数を`target`
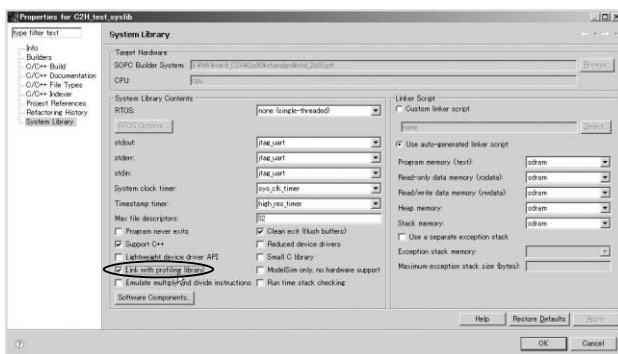
図1　System Library画面
「Link with profiling library」をチェックするとgccの-pgオプションが有効になる．
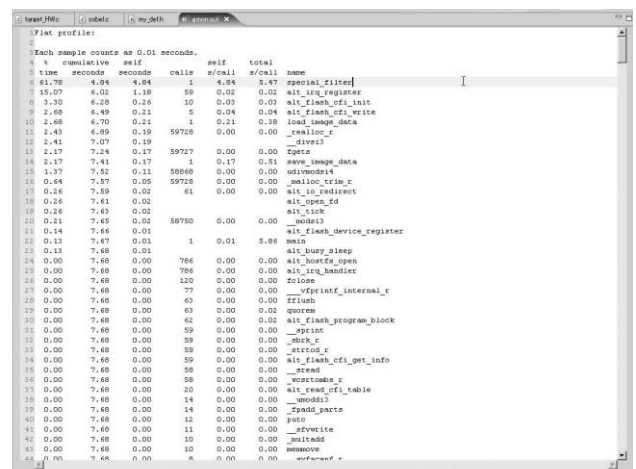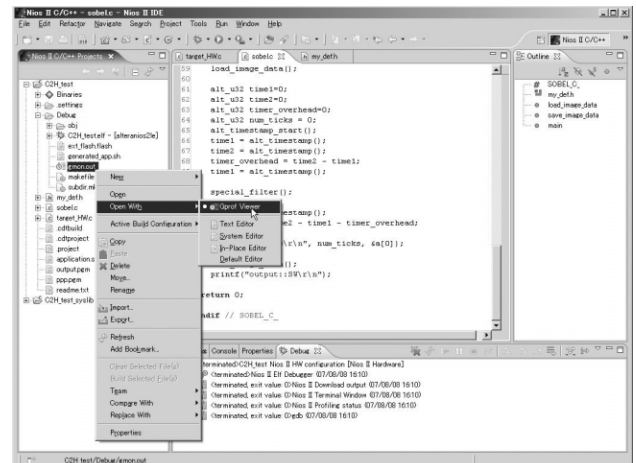
図2　プロファイラ（gmon.out）
モニタリング・ファイルgmon.outは，mainから関数return されるときに生成される．

_HW.cとして独立させました.

● 未サポート記述は使用しない

C2Hは，printfなどの標準関数や再起呼び出し，浮動小数点演算，関数のポインタ，break，continue，goto，__asm("")などの記述をサポートしていません．ハードウェアを合成するためには，未サポート記述が含まれていないかを確認しておく必要があります．

今回の例では，オリジナル・コード内では浮動小数点変数(double)が使用されていました．そこで，double宣言をint宣言に修正しました．

● ハードウェア化に適さない記述のチェック

グローバル変数やスタティック関数を使用すると，ハードウェア合成時に冗長回路を生成してしまいます．ハードウェア化の対象関数でこれらが用いられている場合，引き数として与えるように修正します．

また，C関数のソース・コードが200行以上の場合は，関数の分割を検討します．

今回の例では，複数のグローバル変数が使われていました．これを引き数として与えるため，**リスト1**のように修正しました．

リスト1　複数のグローバル変数を引き数として与える

(**a**) オリジナルのCコード

```
void special_filter(void){
```

(**b**) 修正後のCコード

```
void special_filter( int x_size1, int y_size1,
                image1[256][256], image2[256][256]){
```

● ハードウェア化に向いた記述に変更

より性能の高いハードウェアを合成するために，C2Hには推奨コーディング・スタイルがあります．例えば，1行に複数の演算がある場合は，複数の行に分けて記述します．また，2重ループなどの内側のループでは，サブ関数を使用しないようにします．

ソフトウェア的な最適化では，繰り返し回数の少ないループを外して記述することがあります．しかしC2Hでは，ハードウェア・リソースが増えるので，シンプルなコードにします(**リスト2**)．

リスト2　シンプルなコードにする

(**a**) オリジナルのCコード

```
for(y=1;y<y_size1-1;y++){
  for(x=1;x<x_size1-1;x++){
    pixel_value2=0;
      pixel_value2 = pixel_value2 +
                weight[0][0] * image1[y-1][x-1];
      pixel_value2 = pixel_value2 +
                weight[0][1] * image1[y-1][x];
      pixel_value2 = pixel_value2 +
                weight[0][2] * image1[y-1][x+1];
      pixel_value2 = pixel_value2 +
                weight[1][0] * image1[y][x-1];
      pixel_value2 = pixel_value2 +
                weight[1][1] * image1[y][x];
      pixel_value2 = pixel_value2 +
                weight[1][2] * image1[y][x+1];
      pixel_value2 = pixel_value2 +
                weight[2][0] * image1[y+1][x-1];
      pixel_value2 = pixel_value2 +
                weight[2][1] * image1[y+1][x];
      pixel_value2 = pixel_value2 +
                weight[2][2] * image1[y+1][x+1];
    if(pixel_value2<min2) min2 = pixel_value2;
    if(pixel_value2>max2) max2 = pixel_value2;
}
```

(**b**) 修正後のCコード

```
for(y=1;y<y_size1-1;y++){
  for(x=1;x<x_size1-1;x++){
    pixel_value2=0;
    for(i=-1;i<2;i++){
      for(j=-1;j<2;j++){
        pixel_value2 = pixel_value2 +
                weight[i+1][j+1] * image1[y+i][x+j];
          }
    }
    if(pixel_value2<min2) min2 = pixel_value2;
    if(pixel_value2>max2) max2 = pixel_value2;
}
```

● C2Hに特有の記述を追加

ポインタを使用している場合，ハードウェアではバス・マスタとして合成されます．ここで，複数のポインタ変数の参照する可能性のあるアドレス領域がオーバラップしているとC2Hが判断すると，バス・アクセスが制限されてしまいます．

そこで，ポインタ(バス・マスタ)参照するアドレスが重複しないことがあらかじめ分かっている場合は，__restrict__ という修飾子を使用して，依存関係を制御します．この記述により，バス・マスタのアクセスが制限されなくなります．今回は，**リスト3**のような修正を行いました．

また，#pragma修飾子を使用することで，マスタの接続先を制御できます．例えば，関数名/変数名 をsdramに対してアクセスする場合は，以下のように記述します．

#pragma altera_accelerate connect_

リスト3　ポインタ変数を制御する

(a) オリジナルのCコード

```
int *a;
```

(b) 修正後のCコード

```
int * __restrict__ a;
```

```
variable  special_filter/image1  to
sdram
```

● ハードウェア処理に向いたアルゴリズムに修正

ポインタ(バス・マスタ)は，アドレスのアライメントを行うことで，高効率のデータ転送が期待できるようになります．

また，多重ループを1重化すると，パイプラインの効率が上がります．これは，ループ動作をパイプライン処理に変換するためです(**リスト4**)．

● ハードウェア・リソースを節約する工夫

Nios IIは32ビットのRISCプロセッサです．0～31の範囲でしか変化しないと分かっている変数であっても，int宣言すると，32ビット幅のデータとして合成されてしまいます．取り得る値が0～31であれば5ビットでよいので，残りの27ビットが無駄になってしまいます．このようなとき，ANDでマスクする記述を加えると，使用するハードウェア・リソースを減らせます．

今回は，**リスト5**のような修正を行いました．

3．ハードウェア化の効果を調べる

評価する前段階として，基準になるデータが必要になります．そこで，オリジナルのCソース・コードを基準として，C2Hによるハードウェア化の効果について調べてみます．

比較に当たり，基準としたのは，オリジナルのCソース・コードに対してファイル分離と未サポート記述の修正，適さないコードの確認の3項目の変更を行ったものとします．つまりdouble型の変数をint型にして，ハードウェア化以降と同じ処理を行うソフトウェア・コードを基準にします．

基準となるソフトウェア・コードの実行時間を測定すると，256,305,406サイクルとなりました．

また，同じコードをC2Hでハードウェア化すると，実行時間は25,798,491サイクルでした．いきなり約10倍程度改善の効果が得られました．今回の各段階における結果を**表1**にまとめます．参考までに，チューニングにかかった時間(工数)も示しています．

C2Hが出力したレポートを**図3**に示します．CPLI(Cycles Per Loop Iteration)が高い部分は，まだチューニングの可能性が残されていることを意味します．レポートの右側には，CPLI値を小さくするための修正案が書かれ

リスト4　多重ループを1重化する

(a) オリジナルのCコード

```
for(y=1;y<y_size1-1;y++){
  for(x=1;x<x_size1-1;x++){
    pixel_value2=0;
    for(i=-1;i<2;i++){
       for(j=-1;j<2;j++){
         pixel_value2 = pixel_value2 +
                 weight[i+1][j+1] * image1[y+i][x+j];
               }
    }
    if(pixel_value2<min2) min2 = pixel_value2;
    if(pixel_value2>max2) max2 = pixel_value2;
  }
```

(b) 修正後のCコード

```
for(y=1;y<y_size1-1;  ){
  pixel_value2 = pixel_value2 + weight[i+1][j+1]
                                 * image1[y+i][x+j];
  if(j<1){
    j+=2;
      }else{
       i++;
       j=-1;
       }
  if(i==2){
   i=-1;
      if(x<(x_size1-2)){
        x++;
      }
      else{
       y++;
       x=1;
     }
  if(pixel_value2<min2) min2 = pixel_value2;
  if(pixel_value2>max2) max2 = pixel_value2;
  pixel_value2=0;
 }
}
```

リスト5　未使用ビットをANDでマスクする

(a) オリジナルのCコード

```
uint_8_data = calc_data;
```

(b) 修正後のCコード

```
uint_8_data = calc_data & 0xff;
```

表1　チューニング結果

チューニング内容	ソフトウェア			C2H		
	初期状態	単純実行	C2H向き記述	C2H特有記述	ハードを意識	リソース節約
LE数（換算）	4,628	15,105	11,800	12,295	12,088	11,730
メモリ	571,136	573,184	572,160	573,184	572,160	572,160
DSPブロック	8	48	48	56	56	44
サイクル数	256,305,406	25,798,491	24,419,070	23,679,535	11,860,512	11,862,710
倍率	1.0	9.9	10.5	10.8	21.6	21.6
工数[分]	5	20	15	15	20	10

LE：Logic Element
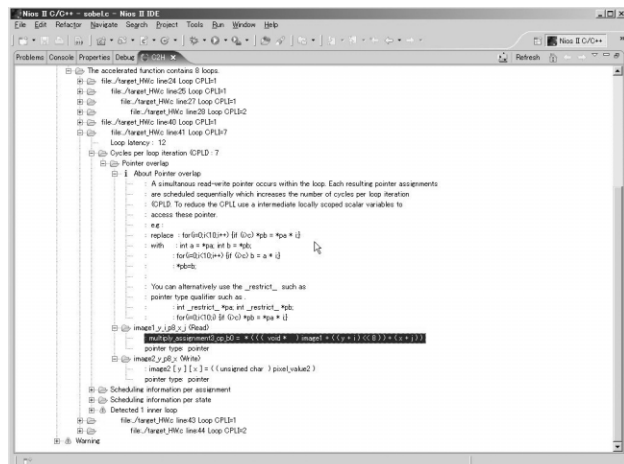
図3　C2Hが出力したレポート
CPLI値と改善策が表示される．

ています．今回は，C2Hを活用する際に一般的なチューニング作業に注目しているので，これらの修正案については対策を施しません．実際の設計で，よりチューニングが求められるときには，このレポート結果を参考にするとよいでしょう．

C2Hは，ソース・コードへの修正をほとんど行わなくても性能の改善が得られますが，相応のハードウェア・リソース（ロジック・エレメント）を必要とします．

グローバル変数を引き数に修正するだけでも，使用するリソース数は減ります．グローバル変数をそのままハードウェア化してしまうと，冗長回路が生成されます．

性能面で一番効果があるのは，パイプライン処理です．すなわち，多重ループの1重化が最も効果的であることが分かります．しかも今回のチューニングの範囲では，CPLI値が1になっていないので，まだ高速化の余地はありそうです．

また配列の記述のしかたが，C2Hで効果を得るためのかぎになっています．今回の例の場合であれば，

- `weight`は配列を使用せずに直接引き数として渡すか，関数の起動時にローカル変数にコピーする．
- `image1`はローカル配列にコピーしてから処理する．

とします．

さらなるチューニングをする場合に，改善が期待できそうな点をまとめます．

- 乗算，除算を使用せずに，加算，減算に変更
- 2^n倍はビット・シフト（<<）を代用
- ライン・バッファの追加で，さらに並列化

参考・引用*文献

(1) Altera社のC2Hのページ，http://www.altera.com/training/sopc-design-ja
http://www.altera.com/training/c2h-fundamemtals-ja
(2) 安居院 猛，長尾智晴；C言語による画像処理入門，昭晃堂，2000年．

いがり・たかひろ
(株)エルセナ

<筆者プロフィール>
猪狩貴寛．Nios Ⅱを含めたシステムの技術サポートを担当．CPUを利用したASIC設計経験後，現在話題のソフト・コア・プロセッサの可能性に魅力を感じ，マルチコア・プロセッサ構成を探索中，現在に至る．